# Program Slicing Based on Sentence Executability

**Salvador V. Cavadini**
Project EVEREST, Institut National de Recherche en Informatique et Automatique (INRIA)
Sophia-Antipolis, 06902, France
Salvador.Cavadini@sophia.inria.fr

**Diego A. Cheda**
DSIC, Universidad Politécnica de Valencia
Valencia, 46022, Spain
dcheda@dsic.upv.es

**Abstract**

We present *point slicing*, a new slicing technique for imperative programs that gives an answer to the question *Which sentences can be executed if sentence* p *is executed?*, very common in program testing, debugging, and understanding tasks and, as far as we know, not directly addressed by other slicing techniques. Point slicing uses a program point as criterion and computes slices by deleting sentences that are proved to be not reachables by executions including the criterion point. We also show how to extend point slicing criterion to a set of program points and how the new technique can be also used to answer to a more precise question: *Which sentences are possibly executed if sentence* p *is executed in a program state satisfying condition* $\phi$*?* Because, minimal point slices are, in general, not computable, we provide definitions of safe approximations for each type of point slice.

**Keywords**: software engineering, program slicing, conditioning, necessary condition for execution.

# 1 Introduction

Since its introduction, program slicing was found to be useful in diverse software engineering domains. Each of these domains use slices with different properties; for this reason several kinds of slices where defined in addition to the original static slice: dynamic slice, quasi-static slice, (pre/post)conditioned slice, amorphous slice, and abstract slice among others.

A very frequent question in program testing, debugging and understanding activities is *Which sentences can be executed if sentence* p *is executed?* Paradoxically, this question, as far as we know, is not directly addressed by any slicing technique.

In this work we present *point slicing*, a new slicing technique where the slicing criterion is a program point and the slice is composed by program sentences that are possibly executed if the sentence of the criterion is executed; this way, point slicing gives a answer to the above question.

We show how point slicing can be extended to use more than one program point as slicing criterion and also how to answer a more precise question: *Which sentences are possibly executed if sentence* p *is executed in a program state satisfying condition* $\phi$*?* All these definitions are given with theirs respective calculi to safely approximate the minimal slices.

This work is organized as follows. The next section provides an overview of the related work. Section 3 introduces much of the definitions that we will use, leaving more specific ones to be introduced when necessary. In section 4 we formally define point slices and provide a calculi to safely approximate them. Section 5 defines two kinds of point slices that use a set of program points as criterion: *weak* and *strong* multipoint slices. Section 6 introduces another extension of point slices: *conditioned* point slices. Next, limitations and future works are discused. Finally, section 8 concludes.

# 2 Related Work

The term program slice was coined by Mark Weiser in his doctoral thesis [14], he also supplied a precise definition and an algorithm for calculate imperative program slices. Weiser stays that programmers, while debugging, divide code into coherent pieces that usually are not lexically adjacent. These pieces are named program slices. In a general sense, *slicing* is a program transformation which preserves some aspects of the semantics of the original program. Usually, the program transformation is sentence deletion.

In the years following the publication of Weiser's work, researchers found program slices useful for other activities different from debugging –program testing, program understanding, program integration, reuse, program maintenance and reverse engineering, among others [15, 10]–. Some of these activities needs different kinds of programs slices that are similar but not the same to the first definition. This fact originated the development of new slices types: closure non executable slices, forward slices, dynamic slices, quasi-static slices, semantic slices, preconditioned slices, etc. [13, 1, 15]

The novel slice definitions presented in this work are very related to preconditioned slices. When slicing with a precondition, the goal is to detect sentences that can be proved to be dead code when the program is executed with inputs satisfying the precondition.

The idea of slicing with a precondition was introduced in [6] –further developed in

```
S ::= S;S
    | skip
    | id := E
    | if (B) then S else S endif
    | do S' loop

S'::= S;S'
    | S
    | break

E ::= an expression
B ::= a Boolean expression
```

Language Grammar

```
1 if (a=b) then
2    if (a=c) then
3       r:='equilateral'
     else
4       r:='isosceles'
     endif
  else
5    if (a=c) then
6       r:='isosceles'
     else
7       if (b=c) then
8          r:='isosceles'
        else
9          r:='scalene'
        endif
     endif
  endif
```

Program *Triangles*

Table 1: Partial grammar of a simple imperative language and an example program

[3], [7] and [2]–. Precondition-induced dead code is detected by symbolic execution of the program. The initial program state satisfying the precondition is propagated to all points in the program and then at each conditional program branch a test is made to check if it is possible to infer the condition's Boolean value from the corresponding program state. The same approach was adopted in [11] with the addition of an automatic theorem prover to aid in the automation of condition checking. An equivalent method was proposed in [5] where strongest postcondition calculus is used to propagate the initial program state through the program. A less precise method for compute precondition slices is introduced in [12]. The technique avoids the use of theorem provers by applying abstract interpretation to reason about the effects of the precondition. This allows full automated preconditioned slicing at the price of precision.

# 3 Preliminary Definitions

We will work with a small imperative programming language (Table 1). Let the sentences of the program be numbered from one to the total number of sentences in the program and the unique exit point of the program be labeled with *end*. The state of a program execution is a set of pairs $(x, y)$ where $x$ is a variable in the program's data space and $y$ its value. Variables not mentioned in the state are considered with an undefined value. Then the pair $\langle \sigma, n \rangle$ means that the sentence $n$ of the program will be executed in a state $\sigma = \{(x_1, y_1), (x_2, y_2), \ldots\}$.

**Definition 3.1** The **execution trace** of program $S$ with input set $\sigma_1$ is the sequence $\tau^S_{\sigma_1} = \langle \sigma_1, 1 \rangle, \ldots, \langle \sigma_m, end \rangle$. Given the predicate $\phi$, $\mathbf{T}^S_{[\phi]}$ denotes the **set of executions traces** produced by executing $S$ with input sets satisfying $\phi$.[1]

[1]Notice that, for the sake of simplicity, non termination is not taken into account.

**Example 3.2** Let $S$ be the program Triangles (Table 1). Then:

- The execution trace for $\{(a,2),(b,2),(c,3)\}$ is:

$$\begin{aligned}\tau^S_{\{(a,2),(b,2),(c,3)\}} \quad = \quad & \langle\{(a,2),(b,2),(c,3)\},1\rangle, \langle\{(a,2),(b,2),(c,3)\},2\rangle,\\ & \langle\{(a,2),(b,2),(c,3)\},4\rangle,\\ & \langle\{(a,2),(b,2),(c,3),(r,'isosceles')\},end\rangle\end{aligned}$$

- The set of execution traces for $[a = 1 \wedge a < b \wedge b = c]$ is:

$$\mathbf{T}^S_{[a=1\wedge a<b\wedge b=c]} \quad = \quad \{\tau^S_{\{(a,1),(b,2),(c,2)\}}, \tau^S_{\{(a,1),(b,3),(c,3)\}}, \tau^S_{\{(a,1),(b,4),(c,4)\}}, \dots\}$$

- If predicate $\phi$ is $false$, then there is no input set that can satisfy it, therefore $\mathbf{T}^S_{[false]} = \emptyset$

**Definition 3.3** We will say that $\tau^S_{\sigma_1}$ **reaches** sentence -program point- $p$, noted $\tau^S_{\sigma_1} \twoheadrightarrow p$, if and only if $\tau^S_{\sigma_1} = \langle\sigma_1,1\rangle,\dots,\langle\sigma_i,p\rangle,\dots,\langle\sigma_m,end\rangle$. If $p$ is not member of any pair of the sequence $\tau^S_{\sigma_1}$, we will write $\tau^S_{\sigma_1} \not\twoheadrightarrow p$.

**Definition 3.4** We will say that $\mathbf{T}^S_{[\phi]}$ **never reaches** $p$, noted $\mathbf{T}^S_{[\phi]} \not\twoheadrightarrow p$, if and only if $\forall\sigma : \big((\sigma:\phi) \Longrightarrow \tau^S_\sigma \not\twoheadrightarrow p\big)$.

**Example 3.5** Let $S$ be the program Triangles (Table 1) then: $\mathbf{T}^S_{[a=b=c]} \twoheadrightarrow 3$, $\mathbf{T}^S_{[a=b=c]} \not\twoheadrightarrow 5$, $\mathbf{T}^S_{[true]} \twoheadrightarrow 1$ and $\mathbf{T}^S_{[false]} \not\twoheadrightarrow 1$.

The *necessary condition for execution* (***nce***) [4] of a program sentence is a condition on the program input set that is true each time the sentence is executed, i.e. if the point -sentence- of interest is reached by the control flow of the program then the initial input set satisfied the ***nce*** of the point. Formally:

**Definition 3.6** The **necessary condition for execution** of sentence $p$ in program $S$, noted $\boldsymbol{nce}(S,p)$, is a precondition that must be true for every input set $\sigma_1$ such that $\tau^S_{\sigma_1} \twoheadrightarrow p$. In other terms: $(\tau^S_{\sigma_1} \twoheadrightarrow p) \Longrightarrow \sigma_1 : \boldsymbol{nce}(S,p)$.

**Example 3.7** Let $S$ be the program Triangles (Table 1), $(a \neq b)$ is a ***nce*** for the program point 6. And $(a \neq b \wedge a = c)$ is a stronger one.

Naturally, the set of inputs satisfying a ***nce*** of a sentence includes the set of inputs satisfying the *necessary and sufficient condition* for the execution of the sentence and this set includes the set of inputs satisfying the *sufficient condition* (a.k.a. *path condition*) for the execution of the sentence. For example, a necessary condition for execution for `b:=1` in the program of Table 2 can be $(b = 0)$ while its path condition is $(c \leqslant 9 \wedge b = 0 \wedge a < 0)$ and its necessary and sufficient condition is $(c \leqslant 9 \wedge b = 0 \wedge a - (10 - c) < 0)$. Appendix A contains a detailed calculi for ***nce***.

# 4 Slicing with a Program Point

A program slice, as defined by Weiser in [14], preserves the behavior of the original program w.r.t. a set of variables at a specified program point. In other words, given a program $S$, a set $V$ of variables and a program point $p$, the slice of $S$ w.r.t. *(V,p)* is *S'*, a subprogram of $S$ such that the executions of $S$ and *S'* are indistinguishable w.r.t. the values of variables in $V$ at the point $p$.

```
if (a<10) then
   x:=0
else
   x:=1
endif;
do
   if (c>9) then
      break
   else
      if ((a<0)&&(b=0)) then
         b := 1
      else
         c := c + 1;
         a := a - 1
      endif
   endif
loop
```

Table 2: Program with a *do-loop* structure

For our pourposes, we will say that a slice or *reduction* is a new program obtained by deleting zero or more sentences from the original program:

**Definition 4.1** A program $S' = s'_1; \ldots; s'_n$ is a **reduction** or a **slice** of $S = s_1; \ldots; s_n$, noted $S' \preceq S$, if and only if $\forall i = [1, n] : (s'_i = s_i) \vee (s'_i = \texttt{skip})$.

We will define *preconditioned slice* [6, 3] as follows:

**Definition 4.2** Given the program $S = s_1; \ldots; s_n$ and the predicate $\phi$, then the **preconditioned slice** of $S$ w.r.t. $\phi$, noted $\lhd(S, \phi)$, is a program $S' = s'_1; \ldots; s'_n$ such that $\forall i = [1, n] : s'_i = \chi(\mathbf{T}^S_{[\phi]} \not\twoheadrightarrow s_i, s_i)$.

where $\chi$ is a function to replace program sentences by `skip` sentences; let $\phi$ be a predicate and $s$ a sentence then function $\chi(\phi, s)$ returns `skip` if $\phi$ holds, $s$ otherwise.

The definition stays that a preconditioned slice of program $S$ w.r.t. the condition $\phi$ is $S'$, a reduction of $S$ where zero or more sentences were changed to `skip` because they can not be part of any execution that starts with an input set satisfying $\phi$.

**Example 4.3** Let $S$ be the program Triangles (Table 1), then $\lhd(S, a \neq c)$ can be obtained by changing to `skip` those sentences in ***italics***.

In this section we introduce *point slicing*, a technique that uses a program point as criterion and computes a slice with the following property:

**Property 4.4** Let $S$ be a program and $S'$ its *point slice* w.r.t. the program point $p$, then: $(\tau^S_x \twoheadrightarrow p) \Longrightarrow \tau^S_x = \tau^{S'}_x$

This is, $S$ and $S'$ are two programs such that if the set $x$ of inputs produces an execution trace $\tau^S$ including sentence $p$, then executiong $S'$ with the same inputs will result in the same execution trace. A definition for this slice is:

**Definition 4.5** Given the program $S = s_1; \ldots; s_n$ and the program point $p$, then the **point slice** of $S$ w.r.t. $p$, noted $\lhd^\bullet(S, p)$, is $S' = s'_1; \ldots; s'_n$ where $\forall i = [1, n] : s'_i = \chi\big(\forall x : (\tau_x^S \twoheadrightarrow p \Longrightarrow \tau_x^S \not\twoheadrightarrow s_i), s_i\big)$.

**Example 4.6** In the Taxation Calculation Program (Table 3), sentences in *italics* must be changed to `skip` in order to obtain a point slice w.r.t. sentence 38. Underlined sentences must be changed to `skip` to get a point slice w.r.t. sentence 4. Notice that ***nce*** for sentence 38 is $((\neg blind) \wedge (\neg married) \wedge (age < 65))$ and that for 4 is $(75 > age >= 65)$.

A simple approximation to this slice can be the set of sentences that result from the union of the set of sentences that reaches –in the control flow graph– the point of interest and the set of sentences that are reached from this point; however, a better –i.e. smaller– approximation can be obtained if we use semantic information of the slice point. We can reinforce the slicing process with the information about the initial conditions that the program inputs must satisfy to reach the point of interest. Unfortunately, the exact computation of these initial conditions is, in general, undecidable, thus we must approximate them.

## 4.1 Safely Approximating Point Slices using Preconditioned Slicing and the Necessary Condition for Execution

The ***nce***, in conjunction with preconditioned slicing, can be used to safely approximate minimal point slices. The approximation is possible because it can be proved that if an execution of program $S$ with inputs satisfying $nce(S, p)$ does not reach the program point $n$ then any execution reaching $p$ do not reach $n$. This allows us to say that $\lhd(S, nce(S, p))$ is a safe approximation of $\lhd^\bullet(S, p)$ or, more formally, that $\lhd^\bullet(S, p) \preceq \lhd(S, nce(S, p))$.

Notice that the path condition of a sentence can not be used to safely approximate point slices. This is because path conditions are too strong and used in preconditioned slicing may led to the deletion of sentences that can be reached by executions that also reach the sentence of interest. As an example of this circumstance, we can see what happens if we compute a preconditioned slice using the path condition of `b:=1` in the program of Table 2, sentence `x:=1` will be deleted –i.e. changed by `skip`– when, actually, there are executions that include both `x:=1` and `b:=1`. Weakest precondition [8] is neither useful for the purpose of answering the above question because, in presence of loops, there is no guarantee that the computed precondition is the weakest one, thus the same problem as with path condition arises.

# 5 Slicing with Two or More Program Points

It is possible to extend point slicing to cope with more than one program point. In this section we define two point slices that use a set of points as criterion:

- *weak multipoint slice* is the slice that encodes all the possible traces that reach any of the given criterion program points, and
- *strong multipoint slice* is the slice that encodes all the possible traces that reach all the given criterion program points.

```
1  if (age>=75) then
2     pers:=5980
3  else if (age>=65) then
4        pers:=5720
5      else pers:=4335
       endif
   endif
6  if (age>=65 &&
      inc>16800) then
7     t:=pers-((inc-16800)/2);
8     if (t>4335) then
9       pers:=t
10    else pers:=4335
      endif
   endif
11 if (blind) then
12    pers:=pers+1380
   endif
13 if (married &&
      age>=75) then
14    pc10=6692
15 else if (married &&
         age>=65) then
16       pc10:=6625
17    else if (married ||
            widow) then
18          pc10:=3470
19       else pc10:=1500
         endif
      endif
   endif
20 if (married && age>=65 &&
      inc>16800) then
21    t:=pc10-((inc-16800)/2);
22    if (t>3470) then
23      pc10:=t
24    else pc10:=3470
      endif
   endif
25 if (inc<=pers) then tax:=0
27 else inc:=inc-pers;
28    if (inc<=pc10) then
29      tax:=inc/10
30    else tax:=pc10/10;
31      inc:=inc-pc10;
32      if (inc<=28000) then
33        tax:=...
        else
34        tax:=...  ;
35        inc:=inc-28000;
36        tax:=...
        endif
      endif
   endif
37 if (!blind && !married &&
      age<65) then
38    code:='L'
39 else if (!blind && married &&
      age<65) then
40    code:='H'
41 else if (!blind && !married &&
      age>=65 && age<75) then
42    code:='P'
43 else if (!blind && married &&
      age>=65 && age<75) then
44    code:='V'
45 else code:='T'
   endif endif endif endif
```

Table 3: UK Income Taxation Calculation Program (borro wed from [9]).

First, we introduce the definitions of union and intersection of program reductions, both will be used to approximate multipoint slices. Consider the program $S = s_1; \ldots; s_n$ and two of its reductions $S_1 = s_{1,1}; \ldots; s_{n,1}$ and $S_2 = s_{1,2}; \ldots; s_{n,2}$ then:

**Definition 5.1** The **union of reductions** is

$$S_1 \cup S_2 = s'_1; \ldots; s'_n \text{ where } \forall i = [1,n] : s'_i = \chi\left((s_{i,1} = \texttt{skip} \wedge s_{i,2} = \texttt{skip}), s_i\right)$$

**Definition 5.2** The **intersection of reductions** is

$$S_1 \cap S_2 = s'_1; \ldots; s'_n \text{ where } \forall i = [1,n] : s'_i = \chi\left((s_{i,1} = \texttt{skip} \vee s_{i,2} = \texttt{skip}), s_i\right)$$

## 5.1 Weak Multipoint Slicing

**Definition 5.3** Given a program $S = s_1; \ldots; s_n$ and a set of program points $Q = \{q_1, \ldots, q_m\}$, the **weak multipoint slice** of $S$, noted $\lhd^{\bullet}_{\mathcal{W}}(S, Q)$, is $s'_1; \ldots; s'_n$ where $\forall i = [1,n] : s'_i = \chi\left(\forall q \in Q,\ \forall x : (\tau^S_x \twoheadrightarrow q \Longrightarrow \tau^S_x \not\twoheadrightarrow s_i), s_i\right)$.

This kind of multipoint slicing computes a reduction such that if an execution trace induced by an input set $x$ in the original program includes one or more sentences of the slicing criterion, then executing the reduction with $x$ will produce the same execution trace. Symbolically: $(\bigvee_{q \in Q} \tau^S_x \twoheadrightarrow q \in Q) \Longrightarrow \tau^S_x = \tau^{\lhd^{\bullet}_{\mathcal{W}}(S,Q)}_x$

We can define two equivalent safe approximations of $\lhd^{\bullet}_{\mathcal{W}}(S, Q)$ as:

$$\bigcup_{q \in Q} \lhd(S, nce(S, q)) \quad = \quad \lhd(S, \bigvee_{q \in Q} nce(S, q))$$

Using properties of ***nce*** it is possible to prove that making the union of the individual point slices of each point in the criterion set results in the same weak multipoint slice obtained from the point slice using the logical disjunction of a ***nce*** of each point.

**Example 5.4** A weak multipoint slice of the program from Table 3 with criterion $Q = \{5, 38\}$ can be obtained by changing to `skip` those sentences that are in ***italics and underlined***.

## 5.2 Strong Multipoint Slicing

**Definition 5.5** Given a program $S = s_1; \ldots; s_n$ and a set of program points $Q = \{q_1, \ldots, q_m\}$, then the **strong multipoint slice** of $S$, noted $\lhd^{\bullet}_{\mathcal{S}}(S, Q)$, is the program $S' = s'_1; \ldots; s'_n$ where $\forall i = [1,n] : s'_i = \chi\left(\exists q \in Q / \forall x : (\tau^S_x \twoheadrightarrow q \Longrightarrow \tau^S_x \not\twoheadrightarrow s_i), s_i\right)$.

This second kind of multipoint slicing computes a reduction such that if an execution trace of the original program induced by an input set $x$ includes all the sentences of the slicing criterion, then the execution of the reduction with $x$ will produce the same execution trace. Formally: $(\bigwedge_{q \in Q} \tau^S_x \twoheadrightarrow q) \Longrightarrow \tau^S_x = \tau^{\lhd^{\bullet}_{\mathcal{S}}(S,Q)}_x$

We can safely approximate the strong multipoint slice of $S$ w.r.t. $Q$ as:

$$\lhd(S, \bigwedge_{q \in Q} nce(S, q)) \preceq \bigcap_{q \in Q} \lhd(S, nce(S, q))$$

Properties of ***nce*** let us prove that the point slice obtained using the logical conjunction of a ***nce*** of each point in the criterion set is smaller than the, also correct, point slice that can be obtained by intersecting the individual point slices of each point.

**Example 5.6** A strong multipoint slice of the program from Table 3 with criterion $Q = \{4, 38\}$ can be obtained by changing to `skip` those sentences that are in *`italics`* `and/or` underlined. This slice correspond to the intersection $\lhd(S, nce(S,4)) \cap \lhd(S, nce(S,38))$. If we use $\lhd\big(S, \bigwedge_{q\in Q} nce(S,q)\big)$ to approximate the point slice, we will get an empty slice because the conjunction between the ***nce*** of 4 and that of 38 is: $\big(((\neg blind) \wedge (\neg married) \wedge (age < 65)) \wedge (75 > age >= 65)\big) \;=\; false$

# 6 Conditioned Point Slicing

In some situations –e.g. while debugging– we are interested in know which are the statements that can be part of execution traces including a given sentence $p$ that is executed in a program state satisfying certain conditions; in other words, we are interested in a slice that can answer to the question *Which sentences are possibly executed if sentence* p *is executed in a program state satisfying condition $\phi$?* Point slicing can be strainforward extended to answer this kind of questions.

**Notation:** We will note $\tau_x^S \twoheadrightarrow end : \phi$ when the execution of $S$ with inputs $x$ ends in a state satisfying predicate $\phi$.

Suppose that $S = s_1; \ldots; s_{i-1}; s_i; \ldots; s_n$ is a program, $\phi$ a predicate and we are only interested in executions of $s_i$ in a state holding $\phi$. We can modify the program to

$$S' = s_1; \ldots; s_{i-1}; \texttt{if } (\phi) \texttt{ then } s_i \texttt{ else skip endif}; \ldots; s_n$$

In this program, $s_i$ is reached only if the execution of $s_{i-1}$ leads to an state satisfying $\phi$, thus:

$$\tau_x^{S'} \twoheadrightarrow \mathtt{s_i} \iff \tau_x^{s_1;\ldots;s_{i-1}} \twoheadrightarrow end : \phi \qquad \text{and} \qquad \tau_x^{S'} \not\twoheadrightarrow \mathtt{s_i} \iff \tau_x^{s_1;\ldots;s_{i-1}} \twoheadrightarrow end : \neg\phi$$

Then if we compute the point slice of $s_i$ in the modified program we will get the set of sentences that can be executed if $s_i$ is executed in a state satisfying $\phi$. It is possible to avoid the program modification by adding the condition as a new term in the point slice criterion and changing appropriately ***nce*** computation (see Appendix A): $nce(\mathtt{S_1; p : S_2}, p, \phi)$ must be defined as $WP_m\ (S_1, \phi)$.

# 7 Limitations and Future Work

Like any other technique that uses statically propagated semantic information, the approximation of point slices through preconditioned slicing loses precision when deals with programs with loop structures. In this particular case, loops decrease precision of both: ***nce*** computation, and preconditioned slicing. A common approach to alleviate this problem is to add user-provided information about loops invariants. This information could be used in ***nce*** computation to get stronger preconditions of programs with loops. The problem with this approach is that is not clear what information must be given by the invariant to be useful in the computation of a correct ***nce***.

A safe approach to get more precise point slices approximations in presence of loops is to use the ***nce***, w.r.t. the point of interest, of the sentence located just after the loop: let be the program $S = S_1; L; S_2$ where $S_1 = s_{1,1}; \ldots; s_{n,1}$, $L = \texttt{do } S_3 \texttt{ loop}$, and $S_2 = s_{1,2}; \ldots; s_{p,2}; \ldots; s_{m,2}$ then:

$$\lhd^{\bullet}(S,p) \;\preceq\; \lhd(S_1; L, nce(S,p)); \lhd(S_2, nce(S_2,p)) \;\preceq\; \lhd(S, nce(S,p))$$

Finally, the definitions introduced in this article are for a simple while language; work is required in order to extend the definition of ***nce*** computation to cope with more complex imperative language features such as function calls and pointers.

# 8 Conclusions

This paper has introduced and formalized *point slicing*, a new technique capable of reduce an imperative program to those sentences that are potentially executed when a given sentence is executed, answering the question: *Which sentences can be executed if sentence* p *is executed?* It has showed that the technique can be extendend to compute slices using a set of points as criterion. In this context, two multipoint slicing definitions are given: *weak* and *strong multipoint slicing*. The paper has also defined *conditioned point slicing*, another extension for the technique. Conditioned point slicing answers a more precise question: *Which sentences are possibly executed if sentence* p *is executed in a program state satisfying condition* $\phi$*?*

The provided safe approximations to point slices and its variants are based in preconditioned slices that can be computed by means of different analysis techniques; this gives great freedom in the choice of the precision/computational cost relation of the method to be used to obtain the point slice approximations.

# References

[1] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[2] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In *Harman, M., Gallagher, K. (Eds.), Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607, Amsterdam, 1999. Elsevier Science.

[3] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Software salvaging based on conditions. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 424–433, Washington, DC, USA, 1994. IEEE Computer Society.

[4] Salvador V. Cavadini and Diego A. Cheda. The necessary condition for execution and its use in program slicing. In *PROLE 2007: VII Jornadas sobre Programación y Lenguajes*, Zaragoza, España, 2007.

[5] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM Press.

[6] Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. Software specialization via symbolic execution. *IEEE Trans. Softw. Eng.*, 17(9):884–899, 1991.

[7] Andrea de Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 9, Washington, DC, USA, 1996. IEEE Computer Society.

[8] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[9] C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward conditioning: A new program specialisation technique and its application to program comprehension. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, pages 89–97, Washington, DC, USA, 2001. IEEE Computer Society.

[10] M. Harman, S. Danicic, Y. Sivagurunathan, and D. Simpson. The next 700 slicing criteria. *2 nd UK workshop on program comprehension (Durham University, UK, July 1996), M. Munro, Ed.*, 1996.

[11] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 138, Washington, DC, USA, 2001. IEEE Computer Society.

[12] B. Kim Jang-Wu Jo I. Chung, B.-M. Chang. Abstract program slicings. In *Twentieth IASTED International Conference on Applied Informatics*. ACTA Press, 2002.

[13] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[14] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

[15] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

# Appendix

# A Computing Necessary Condition for Execution

Although *true* is a valid ***nce*** for any program point, more precise –i.e. stronger– ***nce*** are desireable for preconditioned slicing. In this section we provide a definition for the computation of ***nce***. In this definition, we will use function $WP_m\ (S, \phi)$ that must provide a precondition of $S$ that has the following fundamental property:

$$\tau^S_{\sigma_1} \twoheadrightarrow end : \phi \Longrightarrow \sigma_1 : WP_m\ (S, \phi)$$

where $\tau^S_{\sigma_1} \twoheadrightarrow end : \phi$ says that the execution of $S$ with inputs $\sigma_1$ ends in a state satisfying predicate $\phi$.

The definition of ***nce*** for each basic language construct is the following:[2]

- **Sequence**:

$$\begin{aligned} nce(S_1;\mathtt{p};S_2, p) &= WP_m\ (S_1, true) \\ nce(S_1;Sp, p) &= WP_m\ (S_1, nce(Sp, p)) \end{aligned}$$

[2]**Notation:** we write $Sp$ to mean that the point $p$ is included in the secuence $S$ of sentences.

- **Conditional**:

$$\begin{aligned} nce(\texttt{if (b) then } Sp \texttt{ else } S, p) &= b \wedge nce(S_p, p) \\ nce(\texttt{if (b) then } S \texttt{ else } Sp, p) &= \neg b \wedge nce(S_p, p) \end{aligned}$$

- **Loop**:

$$nce(\texttt{do } Sp \texttt{ loop}, p) = \gamma(mod(S_p), nce(S_p, p))$$

where $mod(S_p)$ is the set of variables that are assigned in $S_p$ and $\gamma$ is a function that takes a set $m$ of variables, and a property $\phi$ and returns a property $\phi_I$ such that: $(\phi \Longrightarrow \phi_I) \wedge (\mathcal{V}(\phi_I) \cap m = \emptyset)$ , where $\mathcal{V}(\phi)$ is the set of the variables referenced in $\phi$. Thus, considering a general property $\phi$ composed with elementary expressions $E$, and logical operators $\wedge, \Longrightarrow, \neg, \vee$ we have that if $\phi$ does not refer to any variable in $w$ then $\gamma(w, \phi) = \phi$. Symbolically: $(\mathcal{V}(\phi) \cap w = \emptyset) \Longrightarrow \gamma(w, \phi) = \phi$

In the other cases:

$$\begin{aligned} \gamma(w, E) &= true \\ \gamma(w, \phi_1 \wedge \phi_2) &= \gamma(w, \phi_1) \wedge \gamma(w, \phi_2) \\ \gamma(w, \phi_1 \vee \phi_2) &= \gamma(w, \phi_1) \vee \gamma(w, \phi_2) \\ \gamma(w, \neg\phi) &= true \\ \gamma(w, \phi_1 \Longrightarrow \phi_2) &= \gamma(w, \neg\phi_1 \vee \phi_2) \end{aligned}$$

### $WP_m$ Computation.

Previously we have defined a way to compute an ***nce*** of a program point using $WP_m$ thus its formalization is given:

- **Skip**: $WP_m\ (\texttt{skip}, \phi) = \phi$
- **Assignment**: $WP_m\ (\texttt{y} := \texttt{exp}, \phi) = \phi_{[y/exp]}$ . Where $\phi_{[y/exp]}$ means that each occurrence of $y$ in $\phi$ was replaced by $exp$.

- **Conditional**: if $S = \texttt{if (b) then } S_1 \texttt{ else } S_2 \texttt{ endif}$, then:

$$WP_m\ (S, \phi) = (c \Longrightarrow WP_m\ (S_1, \phi)) \wedge (\neg c \Longrightarrow WP_m\ (S_2, \phi))$$

- **Break**: $WP_m\ (\texttt{break}, \phi) = false$
- **Loop**: let $L = \texttt{do } S \texttt{ loop}$ and $B(S) = \{b | b = \texttt{break} \wedge S = s_1; \ldots; b; \ldots; s_n\}$

if $B(S) = \emptyset$ then: $WP_m\ (L, \phi) = false$
if $B(S) \neq \emptyset$ then: $WP_m\ (L, \phi) = \bigvee_{s_j \in B(S)} \gamma(mod(S), WP_m\ (s_1; \ldots; s_{j-1}, \phi))$